Announcements

- Midterm #2 grading is underway (and a priority)
  - Endterm exam is non-cumulative - and in class too!
- Next HW assignment is now underway (but…)
  - Indexes and physical DB design
    - Due Tuesday, May 29th (7 days plus one late day)
    - You don’t have all the material yet, ∴ read ahead (!)
- Today’s lecture plan:
  - Finish up B+ tree indexes
  - Quick overview of (static) hash-based indexes
  - Start on physical DB design (use of indexes)
**B+ Tree Deletion (Review)**

- Start at root, find leaf \( L \) where entry belongs.
- Remove the entry.
  - If \( L \) is still at least half-full, done!
  - If \( L \) has only \( d-1 \) entries,
    - Try to **redistribute**, borrowing from **sibling** (adjacent node with same parent as \( L \)).
    - If re-distribution fails, **merge** \( L \) and sibling.
- If merge occurred, must delete search-guiding entry (pointing to \( L \) or sibling) from parent of \( L \).
- Merge could propagate to root, decreasing height.

*Note (see Piazza):* Very cool online B+ tree viz tool available (Ξ)

- [https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html](https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html)
- Only slight differences from our defs (e.g., notice key 13 above)
- Their “Max. Degree” is our \( 2d+1 \) (e.g., limit of 5 ptrs/node above)
Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...

- Deleting 19* is easy.
- Deleting 20* is done with redistribution. Notice how middle key is copied up.

... And Then Deleting 24*

- Must merge.
- Observe “toss” of index entry (on right), and “pull down” of index entry (below).
Example of Non-leaf Redistribution

- New/different example B+ tree is shown below during deletion of 24*
- In contrast to previous example, can redistribute entry from left child of root to right child.

(Note: This shows a temporary illegal tree state w.r.t. d!)

After Redistribution

- Intuitively, entries are redistributed by “pushing through” (or “rotating” if you prefer) the splitting entry in the parent node.
**Bulk Loading of a B+ Tree**

- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- *Bulk Loading* can be done much more efficiently!
- **Initialization:** Sort all data entries, insert pointer to first (leaf) page in a new (root) page.

**Bulk Loading (cont’d.)**

- Index entries for leaf pages always entered into right-most index page just above leaf level. When one fills, it splits. (A split may go up the right-most path to the root.)
- Much faster than repeated inserts!
- Can also control the leaf “fill factor” (%)
A Note on B+ Tree “Order”

- (Mythical!) order (d) concept replaced by physical space criterion in practice (“at least half-full”).
  - Index pages can typically hold many more entries than leaf pages.
  - Variable-sized records and search keys mean that different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (duplicates) can lead to variable-sized data entries in the tree’s leaf pages.

(Page Implementation Details)

Q: What if you were to “open up” a B+ Tree page?
- Control info (e.g., level, # children, free space offset)
- Search key array (with possible on-page indirection for variable-length data, using offsets), or key/data array – for non-leaf vs. leaf pages, respectively
- Child pointer array, where pointer = page id on disk!

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf 1</td>
<td>Ralph</td>
<td>Timothy</td>
</tr>
<tr>
<td>Leaf 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leaf 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### (Leaf Page $I(k)$ Alternatives Revisited)

**Ex:** Emp($eid$, ename, sal, deptid)

**Alternative 1:**
- (records)
- 555: Smith, 18K, 3
- 666: Jones, 90K, 5
- 777: Smith, 23K, 4
- 888: Krishan, 60K, 8

**Alternative 2:**
- (RIDs)
- 444: (P1,4), 12K, 3K
- 555: (P2,1), 18K, 4K
- 666: (P2,2), 23K, 5K
- 777: (P2,3), 3K, 6K
- 888: (P2,4), (P3,1), 7K

**Alternative 3:**
- (RID lists)
- 3K: (P1,1)
- 12K: (P1,4)
- 18K: (P2,1), (P10000,1)
- 23K: (P2,2)
- 60K: (P2,3)

---

### (Leaf Page $I(k)$ Alternatives, cont.)

**Ex:** Emp($eid$, ename, sal, deptid)

**Alternative 1:**
- (records)
- 555: Smith, 18K, 3
- 666: Jones, 90K, 5
- 777: Smith, 23K, 4
- 888: Krishan, 60K, 8

**Alternative 2:**
- (PKs)
- 444: 555, 4439667
- 555: 666, 777
- 666: 888, 888
- 777: 888, 999

**Alternative 3:**
- (PK lists)
- 3K: 111
- 12K: 444
- 18K: 555, 4439667
- 23K: 777
- 60K: 888

**Note:** Must use PKs in secondary indexes when primary index uses Alternative 1!
A Brief Aside: Hash-Based Indexes

- As for any index, 3 alternatives for data entries $k^*$:
  - Data record with key value $k$
  - $<k$, rid of data record with search key value $k$
  - $<k$, list of rids of data records with search key $k$
  - Choice is orthogonal to the indexing technique!
- Hash-based indexes are fast for equality selections. **Cannot** support range searches.
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.

**Static Hashed Indexes**

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- $h(k) \mod N$ = bucket (page) to which data entry with key $k$ belongs. ($N = \#$ of buckets)

<table>
<thead>
<tr>
<th>$k$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h(k) \mod N$</td>
<td>0*, 24*, --</td>
<td>9*, 17*, 81*</td>
<td>26*, 42*, --</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

**Ex:** Using $N = 8$ and $h(k) = k$ (a bad $h(k)$...!)

<table>
<thead>
<tr>
<th>$k$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h(k) \mod N$</td>
<td>0*, 24*, --</td>
<td>9*, 17*, 81*</td>
<td>26*, 42*, --</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Overflow pages</td>
<td>25*, --, --</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Ex:** Find data with key=25 $\rightarrow$ 25*
Static Hashed Indexes (Cont’d.)

- Buckets contain data entries (like for ISAM or B+ trees) – very similar to what we just looked at.
- Hash function works on search key field of record r. Must distribute values over range 0 ... M-1.
  - \( h(key) = (a * key + b) \mod M \) works fairly well.
  - a and b are constants; lots known about how to tune h.
- Long overflow chains can develop and degrade performance. (Analogous to ISAM.)
  - Extendible Hashing and Linear Hashing: More dynamic approaches that address this problem. (Take CS122c!)

Indexing Summary

- Tree-structured indexes are ideal for range searches, also good for equality searches.
- ISAM is a static structure. (Prehistoric B+ Tree!)
  - Only leaf pages modified; overflow pages needed.
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; \( \log_F N \) cost.
  - High fanout \( F \rightarrow \) tree depth rarely more than 3-4.
  - https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
Indexing Summary (Cont’d.)

- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- Most widely used index in DBMS land, and also outside of DBMSs, because of its versatility. Also the most optimized (e.g., for bulk loads, locking, crash recovery, and so on).
- Other database indexes to be aware of:
  - Hash-based (for exact-match queries).
  - R-tree (for spatial indexing and queries).
  - Inverted keyword (for text indexing and queries).